Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №3

по «Алгоритмам и структурам данных»

Базовые задачи (3 задачи)

Выполнил:

Студент группы P3210

Цыпандин Н. П.

Преподаватели:

Косяков М.С.

Тараканов Д. С.

Санкт-Петербург

2022

Задача №1 (I) «Машинки»

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

vector<queue<int>> cars;

class BiHeap {
private:
    vector<int> heap;
    vector<int> ids;

    int left(int parent);

    int right(int parent);

    int parent(int child);

public:
    BiHeap(int n) {
        ids.resize(n, -1);
    }

    void insert(int element);

    void sift_up(int index);

    void sift_down(int index);

    void pop();

    int top();

    int index_by_id(int id);
};

bool cmp(int car1, int car2);

int main() {
    int n, k, p;
    cin >> n >> k >> p;

    cars.resize(n);
    vector<int> queries(p);

    /**
     * Заполним очереди - индексы вхождения для каждой из машин
     */
    for (int i = 0; i < p; ++i) {
        int car;
        cin >> car;
        queries[i] = --car;
        cars[queries[i]].push(i);
    }

    vector<bool> floor(n, false);
    BiHeap heap(n);
    int answer = 0, floor_cnt = 0;

    for (int i = 0; i < p; ++i) {
        const int car = queries[i];
        /**
         * Если машина уже на полу, то просто обновим её позицию в куче
         * (т.к. состояние очереди поменялось)
         */
        if (floor[car]) {
```

```
            cars[car].pop();
            int to_sift = heap.index_by_id(car);
            heap.sift_up(to_sift);
            continue;
        }
        /**
         * Добавим элемент в кучу
         */
        if (floor_cnt < k) {
            cars[car].pop();
            heap.insert(car);
            floor_cnt++;
            floor[car] = true;
        }
        /**
         * Уберем наиболее выгодную машину (top кучи) и добавим машину с запроса
         */
        else {
            const int to_pop = heap.top();
            floor[to_pop] = false;
            heap.pop();

            cars[car].pop();
            heap.insert(car);
            floor[car] = true;
        }
        answer++;
    }
    cout << answer << endl;
    return 0;
}


int BiHeap::parent(int child) {
    if (child == 0 || child >= heap.size())
        return -1;
    return (child - 1) / 2;
}

int BiHeap::left(int parent) {
    int ret = parent * 2 + 1;
    if (ret >= heap.size())
        return -1;
    return ret;
}

int BiHeap::right(int parent) {
    int ret = parent * 2 + 2;
    if (ret >= heap.size())
        return -1;
    return ret;
}

void BiHeap::sift_up(int index) {
    int p = parent(index);
    if (index >= 0 && p >= 0 && cmp(heap[index], heap[p])) {
        swap(ids[heap[index]], ids[heap[p]]);
        swap(heap[index], heap[p]);
        sift_up(p);
    }
}

void BiHeap::sift_down(int index) {
    int l = left(index), r = right(index), mi_i = l;
    if (l < 0)
        return;
    if ((l > 0 && r > 0) && cmp(heap[r], heap[l]))
        mi_i = r;
```

```
    if (cmp(heap[mi_i], heap[index])) {
        swap(ids[heap[mi_i]], ids[heap[index]]);
        swap(heap[mi_i], heap[index]);
        sift_down(mi_i);
    }
}

void BiHeap::insert(int element) {
    heap.push_back(element);
    ids[element] = heap.size() - 1;
    sift_up(heap.size() - 1);
}

void BiHeap::pop() {
    if (heap.empty())
        return;

    heap[0] = heap.back();
    ids[heap.front()] = 0;
    ids[heap.back()] = -1;
    heap.pop_back();
    sift_down(0);
}

int BiHeap::top() {
    if (heap.empty())
        return -1;
    return heap.front();
}

int BiHeap::index_by_id(int id) {
    return ids[id];
}

bool cmp(int car1, int car2) {
    if (!cars[car1].empty() && !cars[car2].empty())
        return cars[car1].front() > cars[car2].front();

    if (cars[car1].empty())
        return true;
    return false;
}
```

Пояснение к примененному алгоритму:

Для начала заполним массив очередей - индексы запросов на каждую из машин. Для решения задачи будем использовать импровизированную бинарную min кучу, которая отвечает на вопрос: “Какую машину стоит убрать с пола?”. Для поддержания состояния кучи, мы будем использовать индексы каждой из машин, и чем дальше последующий запрос, тем выше в куче номер машины, т.к. нам выгодно убирать не актуальные машины. Далее смоделируем пол с помощью этой кучи и пройдемся по всем запросам обновляя состояние.

Асимптотика: O(P * logK)

(Т.к. проходимся по всем запросам, и обновляем значения бинарного дерева, где максимум K элементов (столько машин может быть одновременно на полу)

Задача №2 (J) «Гоблины и очереди»

```
#include <iostream>
#include <deque>

using namespace std;

void process_even(deque<int> *q1, deque<int> *q2);

int main() {
    int n;
    cin >> n;

    deque<int> q1;
    deque<int> q2;
    char operation;
    int number, cnt = 0;

    for (int i = 0; i < n; ++i) {
        cin >> operation;
        if (operation == '-') {
            cout << q1.front() << endl;
            q1.pop_front();
            if (cnt % 2 == 0)
                process_even(&q1, &q2);

            --cnt;
            continue;
        }
        cin >> number;
        if (operation == '+') {
            q2.push_back(number);
            if (cnt % 2 == 0)
                process_even(&q1, &q2);

        } else {
            q2.push_front(number);
            if (cnt % 2 == 0)
                process_even(&q1, &q2);

        }
        ++cnt;
    }
    return 0;
}

void process_even(deque<int> *q1, deque<int> *q2) {
    int mid = (*q2).front();
    (*q2).pop_front();
    (*q1).push_back(mid);
}
```

Пояснение к примененному алгоритму:

Название задачи говорит само за себя. Будем поддерживать состояние двух двусторонних очередей, мысленно разделив её пополам, где начало второй очереди всегда будет хранить центральный элемент. При добавлении в конец, когда нужно, будем передавать один элемент с правой очереди в левую, так же при добавлении в центр.

Асимптотика: O(N)

Задача №3 (L) «Минимум на отрезке»

```
#include <iostream>
#include <cmath>
#include <vector>

#define INF 1e9
using namespace std;

int min_int(int a, int b) {
    if (a < b)
        return a;
    return b;
}

int main() {
    int n, k;
    cin >> n >> k;
    vector<int> arr(n);

    int block_sz, block_cnt;
    block_sz = (int) sqrt(double(n)) + 1;
    block_cnt = n / block_sz;
    if (n % block_sz != 0)
        block_cnt++;

    vector<int> sq(block_cnt, INF);
    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
        sq[i / block_sz] = min_int(arr[i], sq[i / block_sz]);
    }

    for (int i = 0; i < n - k + 1; ++i) {
        int l = i, r = i + k - 1;
        int mi = INF;
        while (l % block_sz != 0 && l <= r) {
            mi = min_int(arr[l], mi);
            ++l;
        }
        while ((l + block_sz - 1) <= r) {
            mi = min_int(sq[l / block_sz], mi);
            l += block_sz;
        }
        while (l <= r) {
            mi = min_int(arr[l], mi);
            ++l;
        }
        cout << mi << ' ';
    }
    return 0;
}
```

Пояснение к примененному алгоритму:

Решим задачу методом sqrt-декомпозиции. Разрежем массив мысленно на sqrt(N) частей, для которых заранее посчитаем минимумы. Далее, когда поступает очередной запрос (l, r), мы будем искать минимум не среди всего отрезка, а всего лишь среди [(r - l) / sqrt(N)] уже посчитанных отрезков и на левом и правом оставшихся концах.

Асимптотика: O(sqrt(N) * (N - K))